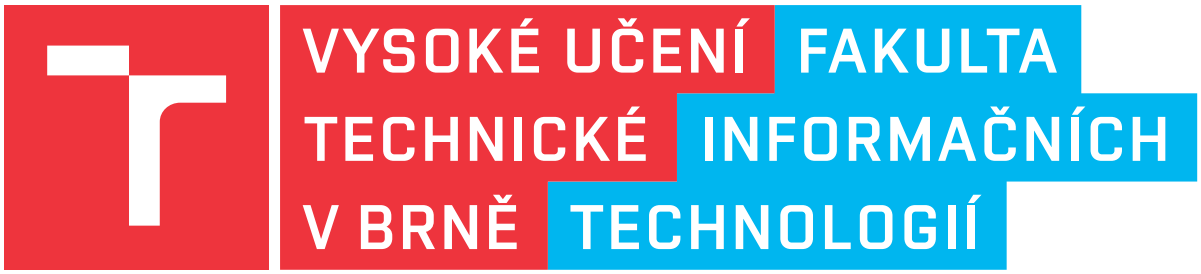
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ



Dokumentácia k projektu do predmetov IFJ a IAL

Implementácia prekladača jazyka IFJ17

Tým 045, varianta I

Peter Grofčík, xgrofc00 (vedúci) 35%
Martin Fujaček, xfujac00 30%
Patrik Krajč, xkrajc17 35%
Rastislav Pôbiš, xpobis00 0%

Obsah

1 Práce v tíme

Zo začiatku sme výber častí nechali v rámci tímu viac-menej voľný a každý si mohol vybrať časť, ktorú by mohol zvládnuť sám, pričom postupne sa úlohy menili, či inak rozdeľovali. Vedúci tímu Peter Grofčík sa venoval prevažne generátoru kódu, pričom sa podieľal aj na tabuľke symbolov. Martin Fujaček pracoval na tabuľke symbolov, dokumentácii a z prevažnej časti sa venoval precedenčnej syntaktickej analýze spolu s Rastislavom Pôbišom, ktorý sa staral aj o testovanie. Patrik Krajč začal s implementáciou lexikálneho a syntaktického analyzátora. Na sémantike pracovali ako Patrik Krajč, tak Peter Grofčík. V záverečných fázach sa práce na jednotlivých moduloch prelínali a každý sa snažil vyriešiť určitý problém bez ohľadu na to, v ktorej časti sa nachádzal. Pri implementácii sme využili aj moduly pre prácu s reťazcami a inštrukčnou páskou, ktoré sú súčasťou jednoduchého demonstračného prekladača.

2 Lexikálny analyzátor

Lexikálny analyzátor je založený na deterministickom konečnom automate, ktorý je uvedený na obrázku 1. Jeho úlohou je rozpoznať jednotlivé lexémy zdrojového kódu a reprezentovať ich ako tokeny. Lexikálny analyzátor je volaný zo syntaktického analyzátora, ktorý ho požiada o ďalšiu lexému. Obsahuje definície jednotlivých stavov konečného automatu, rovnako aj kľúčových a rezervovaných kľúčových slov jazyka IFJ17 založenom na jazyku FreeBASIC. Obrázok konečného automatu sa nachádza na poslednej strane.(7)

3 Syntaktická analýza

3.1 Syntaktický analyzátor

Vstupným bodom nášho prekladača je syntaktický analyzátor, ktorý sme sa rozhodli implementovať metódou rekurzívneho zostupu. Vyhodnocuje prichádzajúce lexémy na základe syntaktických pravidiel, ktoré sú definované LL gramatikou (viď kapitolu 6). Popri syntaktických vykonáva súčasne aj sémantické kontroly. Výrazy sú spracovávané v osobitnom module popísanom nižšie. Až po úspešnom ukončení syntaktickej a sémantickej kontroly sa volá generátor cieľového kódu.

3.2 Precedenčná syntaktická analýza pre výrazy

Syntax výrazov sa vyhodnocuje podľa zadania na základe precedenčnej tabuľky. Tá je v skutočnosti implementovaná ako menšia tabuľka, než je uvedené v tabuľke 1,

keďže niektoré stĺpce sú rovnaké ako iné. Preto každý token mapujeme do tabuľky veľkosti 8×8, kde sú len prvky *Operátor 1* (operátory + a -), *Operátor 2* (operátory * a /), *Operátor 3* (operátor \), *(*, *)*, *Relačné operátory* (=, <>, <, <=, >, >=), *i* (identifikátory, typy integer, double, string) a nakoniec *$*, (špeciálny ukončovací znak). Prichádzajúce tokeny, rovnako ako tokeny najbližšie vrcholu na pracovnom zásobníku precedenčnej analýzy, sú mapované do tabuľky funkciou `group_tokens()`, ktorá podľa zadaného parametru vráti správny index do zmenšenej tabuľky pre konkrétny token. Vstupný reťazec, ktorý analyzujeme v precedenčnej analýze, posiela syntaktický analyzátor v jednosmernom zozname ako prvý parameter vstupnej funkcie `psa()`. Tento zoznam obsahuje stringy jednotlivých tokenov, ich typ a samozrejme ukazovateľ na nasledujúci token v zozname. Ďalšími parametrami sú typ príkazu, v ktorom sa výraz nachádza (napr. priraďovací, návrat z funkcie, cyklus ...), názov premennej (ak sa jedná o priraďovací príkaz, v opačnom prípade voláme s NULL) a názvom funkcie, v ktorej sa práve nachádza. Tieto informácie sú potrebné pre naplnenie inštrukčnej pásky správnymi inštrukciami.

	+	-	*	/	\	(	)	=	<>	<	<=	>	>=	id	int	double	string	$
+	>	>	<	<	<	<	>	>	>	>	>	>	>	<	<	<	<	>
-	>	>	<	<	<	<	>	>	>	>	>	>	>	<	<	<	<	>
*	>	>	>	>	>	<	>	<	<	<	<	<	<	>	>	>	>	>
/	>	>	>	>	>	<	>	<	<	<	<	<	<	>	>	>	>	>
\	>	>	<	<	>	<	>	>	>	>	>	>	>	<	<	<	<	>
(	<	<	<	<	<	<	=	<	<	<	<	<	<	<	<	<	<	
)	>	>	>	>	>		>	>	>	>	>	>	>					>
=	<	<	<	<	<	<	>	>	>	>	>	>	>	<	<	<	<	>
<>	<	<	<	<	<	<	>	>	>	>	>	>	>	<	<	<	<	>
<	<	<	<	<	<	<	>	>	>	>	>	>	>	<	<	<	<	>
<=	<	<	<	<	<	<	>	>	>	>	>	>	>	<	<	<	<	>
>	<	<	<	<	<	<	>	>	>	>	>	>	>	<	<	<	<	>
>=	<	<	<	<	<	<	>	>	>	>	>	>	>	<	<	<	<	>
id	>	>	>	>	>		>	>	>	>	>	>	>					>
int	>	>	>	>	>		>	>	>	>	>	>	>					>
double	>	>	>	>	>		>	>	>	>	>	>	>					>
string	>	>	>	>	>		>	>	>	>	>	>	>					>
$	<	<	<	<	<	<		<	<	<	<	<	<	<	<	<	<	1

Tabulka 1: Tabuľka pre precedenčnú analýzu

4 Generátor kódu

Generátor kódu sa volá po úspešnom ukončení syntaktickej a sémantickej analýzy. Spracúva inštrukčnú pásku a postupne na štandartný výstup tlačí postupnosť inštrukcií, ktoré sa budú interpretovať. Pri implementácii sme využili modul pre prácu s inštrukčnou páskou z jednoduchého demonstračného prekladača, ktorú sme si pre svoje potreby upravili podľa nutnosti. Pomocné tmp premenné pre výsledky porovnaní si generátor rieši sám keďže je jednoznačné že finálne precedenčkou upravený výraz sa pri porovnaniach skladá z dvoch operandov a operátora. Názvy skokov pre interpret IFJ17 sú tvorené názvami operátorov, typu porovnania a samozrejme čísla poradia tmp premennej, ktorá je pri výraze porovnávaná, aby nedošlo k duplicite v prípade rovnakých výrazov.

5 Tabuľka symbolov

Tabuľku symbolov sme implementovali ako binárny vyhľadávací strom podľa postupov preberaných v predmete IAL. Tabuľka obsahuje odkazy na binárne stromy pre každú funkciu v programe. Každý uzol obsahuje niekoľko prvkov. Kľúčom, podľa ktorého sa vyhľadáva, je samotný názov premennej resp. funkcie, ktorý je poľom znakov `name`. Ďalej obsahuje celočíselnú hodnotu `type`, ktorá určuje dátový typ položky. Nesmú chýbať ukazovatele na ľavý, resp. pravý podstrom `LPtr` a `RPtr`, a tiež ukazatele na parametre funkcií, samozrejme s typom daného parametru.

6 LL Gramatika

(1) PROG → DECFLIST DEFFLIST scope STLIST end scope

(2) DECFLIST → ε
(3) DECFLIST → declare FUN DECFLIST

(4) FUN → function funid (PARLIST) as TYPE *EOL*

(5) DEFFLIST → ε
(6) DEFFLIST → FUN STLIST end function *EOL* DEFFLIST

(7) PARLIST → ε
(8) PARLIST → id as TYPE PARLIST1

(9) PARLIST1 → ε
(10) PARLIST1 → , id as TYPE PARLIST1

(11) STLIST → ε
(12) STLIST → STAT *EOL* STLIST

(13) STAT → ε
(14) STAT → DECV
(15) STAT → DECV = RVALUE
(16) STAT → id = RVALUE
(17) STAT → input id
(18) STAT → print E ; PRINT1
(19) STAT → if E then *EOL* STLIST else *EOL* STLIST end if
(20) STAT → do while E *EOL* STLIST loop
(21) STAT → return E

(22) PRINT1 → E ; PRINT1
(23) PRINT1 → ε

(24) RVALUE → E
(25) RVALUE → funid (TERMLIST)

(26) DECV → dim id as TYPE

(27) TYPE → integer
(28) TYPE → double
(29) TYPE → string

(30) TERMLIST → ε
(31) TERMLIST → TERM , TERMLIST

(32) TERM → ε
(33) TERM → TYPE
(34) TERM → id

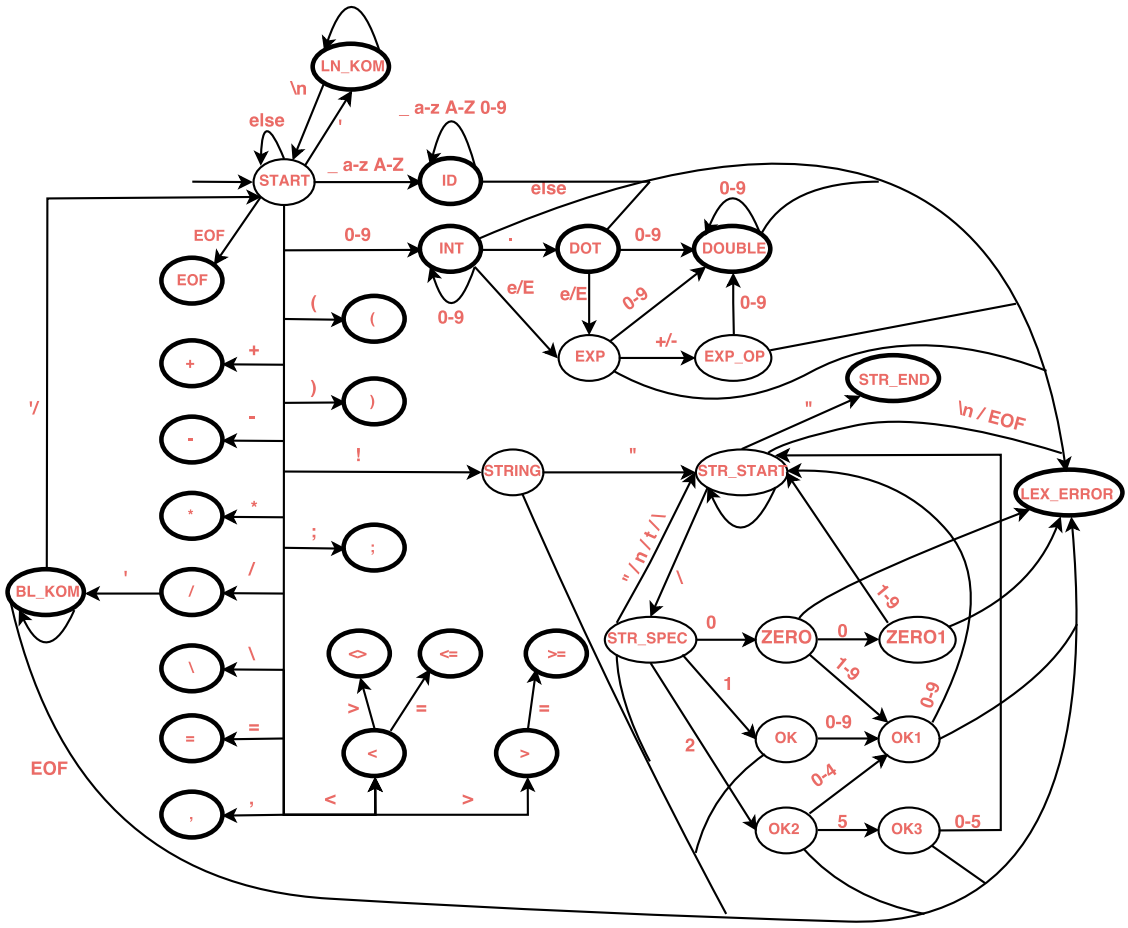
E → (E)
E → E * E
E → E / E
E → E \E
E → E + E

E → E - E
E → E = E
E → E <> E
E → E < E
E → E <= E
E → E > E
E → E >= E
E → id
E → int
E → double
E → string

7 LL tabul'ka

	scope	end	declare	function	funid	(	)	as	eol	id	$	=	input	print	E	;	if	then	else	do	while	loop	return	dim	integer	double	string	$
PROG	1		1	1																								35
DECFLIST	2		3	2																								
FUN	35	35	35	4					35	35			35	35			35			35			35	35				
DEFFLIST	5			6																								
PARLIST							7			8																		
PARLIST1							9				10																	
STLIST		11							12	12			12	12			12		11	12		11	12	12				
STAT	1								13	16			17	18			19			20			21	15				
PRINT1									23						22													
RVALUE					25				35						24													
DECV									35			35												26				
TYPE							35		35		35	35													27	28	29	
TERMLIST							30			31	31														31	31	31	
TERM										34	32														33	33	33	

Tabulka 2: LL Tabul'ka



Obrázek 1: Konečný automat lexikálneho analyzátora